

## CSE 141L | Lab Writeup 1

---

### 1 Introduction

For this lab, our goal is to design the instruction set architecture for a 9-bit processor. Since we are only given a 9 bit processor and we need to be able to manipulate up to 256 memory locations, we decided to allocate our 9 bits as follows: 4 bits to indicate one register operand, 4 bit **opcode** to indicate which operation we will be executing this cycle, and a single bit to indicate the type of the instruction.

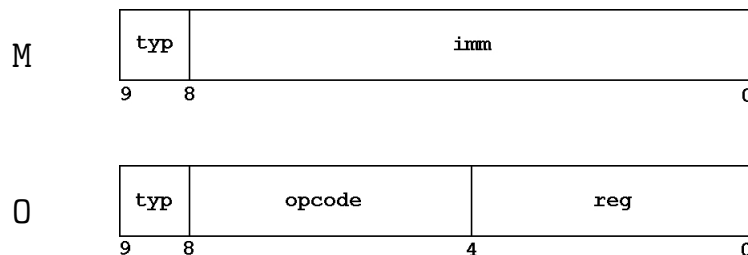
We decided to include this single bit instruction type indicator due to the constraint of having only 9 bits while needing to represent 256 memory locations (which requires 8 bits). Thus, we came up with 2 instruction types as indicated by the single instruction type bit. The first instruction type is the **M-type** which is dedicated only to reading the 256 memory locations. The second **0-type** is used for other register based operations not dealing with memory.

In conclusion, our goal for this lab was to design a preliminary ISA inspired by the Assumulator and the Mem-Reg ISAs. Our end product for this lab supports 16 registers, 16 instructions, and reading data from 256 memory locations.

We call our ISA the **Encryption-Decryption-Message** processor, or EnDMe for short

### 2 Instruction Formats

We introduce 2 instruction formats: the **M-type** and the **0-type**. The **M-type** is a dedicated instruction for memory access brought about due to the constraints of having to access 8-bits worth of memory spaces with only 9-bits available. The **0-type** indicates either a register operation instruction or an instruction to manipulate the accumulator register. The formats are structured as illustrated below:



### 3 Operations

| Op                       | Type | Opcode | Syntax      | Operation  | Details   |
|--------------------------|------|--------|-------------|--|---|
| Data Transfer Operations |      |        |             |  |   |
| save                     | M    | ----   | save #imm   | \$acc = #imm   | Loads immediate into accumulator  |
| store                    | 0    | 0000   | store \$reg | \$reg = \$acc  | Loads accumulator content into \$reg  |
| lb                       | 0    | 0001   | lb \$reg    | \$acc = M[\$reg]                                     | Loads a byte from the address in \$reg into \$acc   |
| sb                       | 0    | 0010   | sb \$reg    | M[\$reg] = \$acc                                     | Stores accumulator data into address in \$reg   |
| put                      | 0    | 0011   | put \$reg   | \$acc = \$reg  | Loads contents of \$reg into accumulator  |
| Branching Operations     |      |        |             |  |   |
| btr                      | 0    | 0100   | btr         | if(\$acc == 1)<br>\$pc = {\$pc[15:8], \$dst}         | Jumps to label/address in \$dst given \$acc holds 1   |
| jmp                      | 0    | 0101   | jmp         | \$pc = {\$pc[15:8], \$dst}                           | Unconditionally jump to the label/address in \$dst  |
| Arithmetic Operations    |      |        |             |  |   |
| add                      | 0    | 0110   | add \$reg   | \$acc = \$reg + \$acc                                | Sums the contents of accumulator with \$reg   |
| sub                      | 0    | 0111   | sub \$reg   | \$acc = \$reg - \$acc                                | Subtracts contents of accumulator from \$reg  |
| Bit-Oriented Operations  |      |        |             |  |   |
| and                      | 0    | 1000   | and \$reg   | \$acc = \$acc & \$reg                                | Does bitwise AND and of \$reg with accumulator  |
| xor                      | 0    | 1001   | xor \$reg   | \$acc = \$acc $\oplus$ \$reg                         | Does exclusive or of \$reg with accumulator   |
| sfl                      | 0    | 1010   | sfl \$reg   | \$acc = \$acc << \$reg                               | Shifts contents of accumulator left by \$reg bits   |
| sfr                      | 0    | 1011   | sfr \$reg   | \$acc = \$acc >> \$reg                               | Shifts contents of accumulator right by \$reg bits  |
| Logic Operations         |      |        |             |  |   |
| cmp                      | 0    | 1100   | cmp \$reg   | if(\$reg == \$acc)<br>\$acc = 1<br>else<br>\$acc = 0 | Checks the contents of \$reg and the accumulator for equality and sets accumulator to 1 or 0 accordingly                        |
| gtr                      | 0    | 1101   | gtr \$reg   | if(\$reg > \$acc)<br>\$acc = 1<br>else<br>\$acc = 0  | Compares the contents of \$reg against the accumulator and accordingly sets accumulator to 1 or 0 depending on which is greater |

## 4 Internal Operands

We use 4 bits to indicate 16 registers. Aside from the accumulator register, we define 2 special registers. The last register is designated as a destination register that should not be touched by programs aside for storing a jump location prior to a **btr** instruction. This register is for the exclusive use of the Branch on True instruction **btr**; this is **\$dst**. Finally, the second-to-last register is our designated program counter **\$pc**. Aside from these special registers the remaining registers are general purpose registers.

| Number | Name              | Reg Code  | Purpose   |
|--------|-------------------|-----------|---|
| 0      | <b>\$acc</b>      | 0000      | Accumulator, an implied destination or operand register for most instructions |
| 1-13   | <b>\$r1-\$r13</b> | 0001-1101 | General purpose registers (Non-preserved)                                     |
| 14     | <b>\$pc</b>       | 1110      | Program counter   |
| 15     | <b>\$dst</b>      | 1111      | Branch on true dedicated destination register                                 |

## 5 Control Flow

We have two branching instructions, namely *branch on true* **btr** and *jump* **jmp**. Preliminarily we went with absolute addressing for these instructions. Both instructions make use of an explicit address to complete the jump. However, once we completed the assembly code, we found that absolute addressing would only give us access to the first 256 lines of our code. Therefore, we plan to change the control flow of our ISA so that instead of absolute addressing, our ISA uses relative addressing using jump offsets instead of exact addresses in the near future. Using addition to calculate the memory address much like in the MIPS architecture, this will allow us to jump forwards or backwards a range of 200+ lines which we deem is more than enough for our custom assembly code.

## 6 Addressing Modes

We chose direct addressing for our purposes due to the fact that we only need to be able to access 256 memory spaces which can be represented using 8-bits. Every time we want to read or write from a memory location, we first save its index to accumulator, followed by a load word or store word depending on whether we are accessing or storing data into memory. For example, let's say we want to get the value in memory location 128 and want to save it into memory location 129:

```
save #128      #save 128 to accumulator
store r2       #store it to r2
save #129      #save 129 to accumulator
store r3       #store it to r3
lw r2          #load M[128] into accumulator
sw r3          #store the data into M[129]
```

## 7 Architecture Type and Example

We originally intended to base our ISA on the design of an accumulator but due to the addition of several registers, our custom ISA became somewhat of a hybrid between an accumulator architecture and mem-reg architecture.

The following is a simple example of how either M- or O-type instructions should look in machine code:

| Instruction | Machine Code |                 |         |
|-------------|--------------|-----------------|---------|
| save #65    | TYPE         | IMM             |         |
|             | 1            | 0 1 0 0 0 0 0 1 |         |
| xor \$r2    | TYPE         | OPCODE          | REG     |
|             | 0            | 1 0 0 1         | 0 0 1 0 |

## 8 Tentative Block Diagram

